



부산대학교

PUSAN NATIONAL UNIVERSITY

Open Addressing

Finding the Next Open Slot

Load, Probing, Deletion, and Python dict

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Load Factor and Resizing

표가 차는 정도와 크기 재할당(resize)이 탐색 비용에 미치는 영향을 설명한다

load factor

정의

load factor

길이 m 인 표에 원소가 n 개일 때 $\alpha = n/m$ 으로 둔다. α 가 커질수록 빈 칸이 줄고 충돌 가능성이 커진다. λ 로 쓰기도 한다.

표를 늘려야 하는 신호

관찰	영향
α 가 계속 증가	충돌과 탐색 길이가 늘어남
빈 칸이 거의 없음	다음 저장 위치를 찾기 어려움
특정 기준에 도달	더 큰 표로 재할당

Caution 재할당 기준은 구현마다 다르므로 한 기준값을 모든 해시테이블에 적용하지 않는다.

재할당은 표 길이를 바꾼다



원래 표의 칸 순서대로 14, 6, 10을 새 표에서 $k \bmod 8$ 부터 다시 조사해 넣으면 위치가 달라지고 빈 칸이 다시 생긴다.

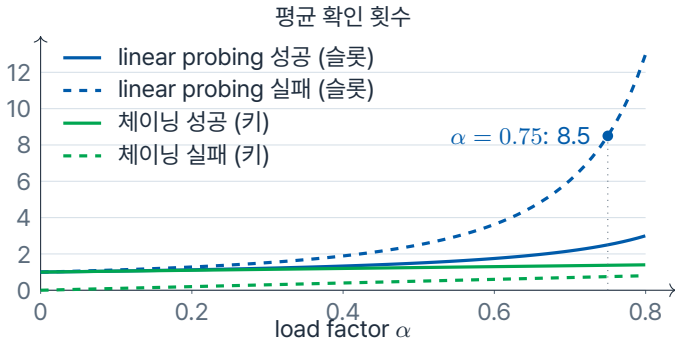
길이가 바뀌면 인덱스도 바뀐다

나머지로 인덱스를 정했다면 m 이 달라질 때 해시 인덱스를 다시 계산해야 한다.

키 k	$k \bmod 4$	$k \bmod 8$
6	2	6
10	2	2
14	2	6

Caution 기존 배열 위치를 그대로 복사하지 말고 모든 키를 새 표에 다시 넣는다.

load factor에 따른 평균 확인 횟수



성공은 키가 있을 때, 실패는 없을 때다. linear probing은 슬롯을 조사하며 실패 시 마지막 빈 칸도 센다. 체이닝은 키 비교를 센다.

식으로 계산한 평균 확인 횟수

키가 고르게 분산된 경우의 근사식이다. linear probing은 슬롯 조사, 체이닝은 키 비교를 센다. 성공은 키가 있을 때, 실패는 없을 때다.

방법과 결과	근사식	$\alpha = 0.5$	$\alpha = 0.75$	$\alpha = 0.9$
linear probing 성공	$\frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$	1.5	2.5	5.5
linear probing 실패	$\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$	2.5	8.5	50.5
체이닝 성공	$1 + \alpha/2$	1.25	1.375	1.45
체이닝 실패	α	0.5	0.75	0.9

α 가 1에 가까워지면 linear probing의 실패 탐색이 급격히 길어진다. 재할당으로 α 를 1보다 충분히 작게 유지하면 탐색은 기대 $O(1)$ 이다.

Checkpoint

Load Factor and Resizing

- 원소 12개를 길이 16인 표에 저장했을 때 α 는 얼마인가?
- 표 길이가 바뀌면 기존 키의 인덱스를 다시 계산해야 하는 이유는 무엇인가?
- $O(n)$ 재할당이 있어도 삽입의 상각 비용이 $O(1)$ 인 이유는 무엇인가?

Linear Probing

충돌하면 다음 칸을 순서대로 확인하는 linear probing을 수행한다

정의

개방주소(open addressing)

버킷마다 리스트를 두지 않고 배열의 다른 빈 칸을 찾아 저장하는 충돌 처리. 배열 한 칸에는 활성 항목을 하나만 두고 정해진 순서로 다음 후보 칸을 확인한다.

한 칸에는 한 항목만 저장한다

배열 칸	가능한 상태
빈 칸	아직 어떤 항목도 저장하지 않음
활성 칸	키와 값 한 쌍을 저장
지운 칸	삭제 흔적을 남김

충돌한 항목은 같은 칸 아래가 아니라 다른 배열 인덱스에 놓인다.

정의

linear probing

충돌한 뒤 인덱스를 한 칸씩 늘려 빈 칸이나 키를 찾는 방법. 길이 m 인 표에서 $h(k)$, $h(k) + 1$, $h(k) + 2, \dots$ 를 m 으로 나눈 나머지 위치를 본다.

충돌한 키는 다음 빈 칸에 놓인다

44와 55는 인덱스 0에서 충돌해 각각 인덱스 1과 2로 이동한다.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

20은 인덱스 9에서 시작해 10, 0, 1, 2를 지나 인덱스 3에 놓인다.

삽입 과정

키	시작 인덱스	확인한 인덱스	저장 인덱스
77	0	0	0
44	0	0, 1	1
55	0	0, 1, 2	2
20	9	9, 10, 0, 1, 2, 3	3

Adapted from Miller and Ranum (2013), Collision Resolution.

탐색도 같은 순서를 따라간다

저장할 때 이동했다면 찾을 때도 그 이동 경로를 재현해야 한다.

만난 상태	탐색 동작
같은 키	값을 반환하고 종료
다른 키	다음 조사 인덱스로 이동
처음부터 빈 칸	키가 없다고 판단하고 종료
지운 칸	다음 조사 인덱스로 이동

배열 끝에서는 인덱스 0으로 돌아간다

$$p_i(k) = (h(k) + i) \bmod m, \quad i = 0, 1, 2, \dots$$

$m = 11$ 이고 $h(k) = 9$ 이면 9, 10, 0, 1, 2, ... 순서로 확인하며, 나머지 연산이 배열 끝을 처음과 연결한다

탐색 종료 조건

종료 조건	판단
같은 키를 만남	성공
처음부터 빈 칸을 만남	실패
표의 모든 후보를 확인	실패

Caution 다른 키를 만났다는 이유만으로 실패라고 판단하면 안 된다.

탐색 코드로 보는 종료 조건

같은 키면 값을 반환하고, 빈 칸을 만나 반복이 끝나거나 시작 칸으로 돌아오면 `None`을 반환한다.

```
1 def get(self, key):
2     start_slot = self.hash_function(key, len(self.slots))
3     position = start_slot
4     while self.slots[position] is not None:
5         if self.slots[position] == key:
6             return self.data[position]
7         else:
8             position = self.rehash(position, len(self.slots))
9             if position == start_slot:
10                 return None
11 def rehash(self, old_hash, size):
12     return (old_hash + 1) % size
```

`rehash`는 다음 조사 인덱스를 계산하는 메서드이고 표 길이를 바꾸는 재할당과 다르다. 이 코드는 지운 칸을 따로 처리하지 않는다.

Checkpoint

Linear Probing

- 길이 11인 표에서 시작 인덱스가 9일 때 처음 여섯 조사 인덱스는 무엇인가?
- 충돌한 키를 찾을 때 저장과 같은 조사 순서가 필요한 이유는 무엇인가?
- 탐색이 처음부터 빈 칸을 만나면 왜 실패로 끝낼 수 있는가?

Probe Sequences

linear probing에서 클러스터링이 생기는 이유를 설명하고 다른 조사 순서와 비교한다

정의

클러스터링(clustering)

linear probing에서 활성 칸이 연속된 긴 구간을 이루는 현상이다. 연속 구간이 길면 탐색에서 더 많은 칸을 확인하게 된다.

충돌이 연속한 칸을 만든다

인덱스 0에서 충돌한 77, 44, 55가 0, 1, 2를 연속으로 채워 이후 그 근처에서 시작하는 탐색을 길게 만든다.

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

77, 44, 55가 만든 클러스터링 구간을 20도 지나야 한다.

클러스터링은 스스로 자란다

상태	결과
연속한 칸이 짧음	빈 칸을 빨리 만남
연속한 칸이 김	많은 칸을 지나 다음 빈 칸을 찾음
새 항목이 뒤에 붙음	연속 구간이 한 칸 더 길어짐

load factor가 같아도 칸의 분포에 따라 조사 수는 달라진다.

조사 간격을 바꿀 수 있다

1칸 대신 제곱이나 키별 값을 간격으로 쓸 수 있다.

방식	번째 후보	이동 모양
linear probing	$h(k) + i$	1칸씩
quadratic probing	$h(k) + i^2$	1, 4, 9, ...
double hashing	$h_1(k) + i h_2(k)$	키별 간격

방법 비교: quadratic probing은 제공만큼 이동한다

충돌하면 $h(k)$ 에서 1, 4, 9, ... 떨어진 칸을 본다. 55는 0, 1, 4, 9, 5를 지나 인덱스 3에 놓인다.

0	1	2	3	4	5	6	7	8	9	10
77	44	20	55	26	93	17	None	None	31	54

linear probing과 다른 위치에 20과 55가 배치된다.

방법 비교: double hashing은 키별 간격을 만든다

$$p_i(k) = (h_1(k) + i h_2(k)) \bmod m$$

$h_2(k)$ 는 0이 아니고 표 길이 m 과 서로소여야 모든 칸을 방문할 수 있다(예: m 이 소수이고 $1 \leq h_2(k) \leq m-1$). 서로 다른 키가 같은 시작 인덱스를 가져도 두 번째 해시 값이 다르면 경로가 달라진다.

Checkpoint

Probe Sequences

- linear probing에서 클러스터링이 스스로 길어지는 이유는 무엇인가?
- quadratic probing의 i 번째 후보 인덱스는 어떻게 계산하는가?
- double hashing에서 두 번째 해시 값 $h_2(k)$ 가 0이면 왜 안 되는가?

Deletion and Tombstones

삭제 뒤에도 기존 조사 경로를 보존하는 tombstone을 사용한다

삭제한 칸을 그냥 비우면 안 된다

충돌로 뒤에 놓인 키까지 가는 조사 경로가 중간에서 끊긴다.

인덱스	0	1	2	3	4
저장 $h(k)$	77	44	55	20	빈 칸
	0	0	0	9	

인덱스 1의 44를 빈 칸으로 바꾸면 55를 찾는 탐색이 인덱스 1에서 잘못 끝난다

빈 칸은 탐색 종료 신호다

칸의 뜻	탐색에서의 판단
처음부터 빈 칸	이 뒤에는 같은 경로로 저장된 키가 없음
삭제로 빈 칸	이 뒤에 충돌로 밀린 키가 있을 수 있음

두 상황을 같은 빈 칸으로 표시하면 탐색이 둘을 구분하지 못한다.

정의

tombstone

항목은 삭제됐지만 조사 경로가 이어진다는 사실을 남기는 특별한 상태. 탐색은 tombstone을 지나가고 삽입은 그 위치를 다시 사용할 수 있다.

Source: OpenDSA (n.d.), Deletion.

탐색은 tombstone을 지나간다

인덱스	0	1	2	3
상태	77	tombstone	55	20

$h(55) = 0$ 이면 인덱스 0의 다른 키 77과 인덱스 1의 tombstone을 지나 인덱스 2에서 55를 찾는다

삽입은 tombstone을 재사용할 수 있다

넣을 키가 이미 있는지 확인하면서 처음 만난 tombstone 위치를 기억한다.

만난 상태	삽입 동작
tombstone	후보 위치로 기억하고 계속 탐색
같은 키	기존 값을 갱신
처음부터 빈 칸	기억한 tombstone 또는 현재 칸에 저장

tombstone도 조사 비용을 늘린다

활성 원소가 적어도 tombstone이 많으면 탐색은 많은 칸을 지나야 한다.

세는 값	용도
활성 원소 수 n	실제 Map 크기
활성 칸과 tombstone 수 q	조사 경로에 포함될 수 있는 칸
표 길이 m	$m \geq 2q$ 를 유지하도록 재할당

새 표로 옮기면 tombstone이 사라진다

단계	동작
새 배열 생성	필요한 길이로 빈 표를 만들
활성 항목 순회	각 키의 조사를 새 표에서 다시 수행
tombstone 제외	삭제 흔적은 복사하지 않음

재할당은 표 길이를 바꾸면서 삭제 흔적도 정리한다.

Checkpoint

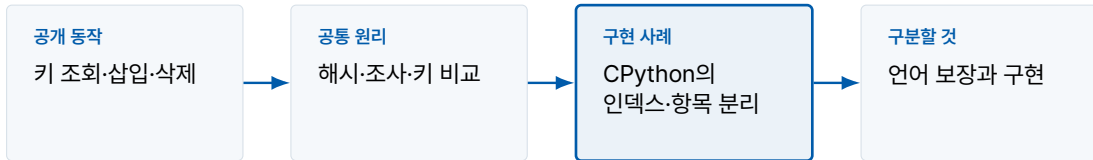
Deletion and Tombstones

- 삭제한 칸을 처음부터 빈 칸으로 바꾸면 어떤 탐색 오류가 생기는가?
- 탐색과 삽입은 tombstone을 각각 어떻게 처리하는가?
- 활성 원소 수와 별도로 tombstone까지 세어야 하는 이유는 무엇인가?

Python dict

Python dict의 공개 동작을 확인하고 CPython 내부 표를 구현 사례로 구분한다

dict는 해시테이블 원리를 확인하는 구현 사례다



CPython의 표 구조는 dict 동작을 설명하는 한 구현이며 Python 언어가 이 내부 구조를 보장하지는 않는다

dict 키는 hashable해야 한다

예	키 사용	이유
<code>int, str</code>	가능	hashable
hashable 원소의 <code>tuple</code>	가능	내용이 바뀌지 않음
<code>list, dict</code>	불가	내용이 바뀔 수 있음

Source: Python 3.14 documentation (n.d.), Mapping Types, dict.

같다고 비교되는 키는 같은 항목을 가리킨다

Python에서는 1, 1.0, True가 같다고 비교된다.

```
d = {1: "first"}  
d[1.0] = "updated"  
len(d)    →    1  
d[True]    →    "updated"
```

해시 값만 보지 않고 키의 동등성 비교까지 해야 Map의 의미가 보존된다.

CPython은 인덱스 테이블과 키·값 항목을 나누어 저장한다



인덱스 테이블에는 빈 칸, 활성 항목의 번호, tombstone을 두고 실제 키와 값은 항목에 보관한다

dict 탐색도 키를 확인한다

단계	개념적 동작
1	키의 해시 값을 계산
2	조사 순서에 따라 인덱스 테이블을 확인
3	후보 항목의 해시와 키 동등성을 확인
4	같은 키면 값을 반환

Source: CPython (n.d.), Objects/dictobject.c.

삭제 뒤에는 tombstone이 경로를 보존한다

tombstone을 곧바로 처음부터 빈 칸으로 되돌리면 충돌로 뒤에 놓인 항목을 찾지 못할 수 있다.

상태	탐색 처리
활성	키를 비교
tombstone	다음 후보로 계속
처음부터 빈 칸	키가 없다고 판단

Source: CPython (n.d.), Objects/dictobject.c, Dummy slot comments.

dict는 삽입 순서를 보존한다

```
d = {"a": 1, "b": 2, "c": 3}
```

```
d["b"] = 20 → 기존 위치 유지
```

```
del d["b"]
```

```
d["b"] = 2 → 마지막에 다시 삽입
```

Caution 삽입 순서 보존은 정렬된 키 순서를 뜻하지 않는다.

Checkpoint

Python dict

- `list`가 `dict` 키가 될 수 없는 이유는 무엇인가?
- CPython의 tombstone은 어떤 조사 경로를 보존하는가?
- `dict`의 삽입 순서 보존과 키 정렬은 어떻게 다른가?

How Does an Open Slot Stay Findable?

Next Lecture: Tree Terminology

- 루트, 부모 노드, 자식 노드, 단말 노드
- 깊이와 높이를 세는 법
- 부분 트리로 보는 재귀 구조